School of Data and Computer Science, Sun Yat-sen University

E03 Othello Game ($\alpha - \beta$ pruning)

16337102 Zilin Huang

September 21, 2018

Contents

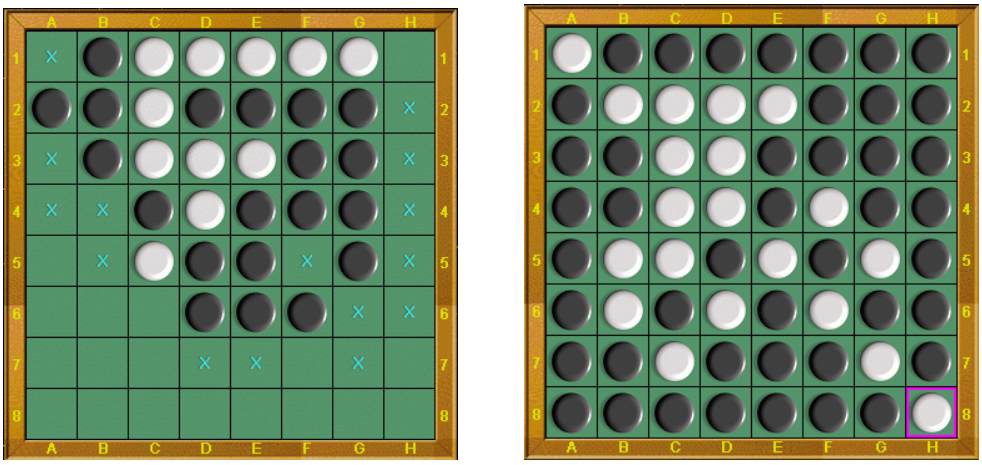


Figure 1: Othello Game

1 Othello

Othello (or Reversi) is a strategy board game for two players, played on an 8×8 uncheckered board. There are sixty-four identical game pieces called disks (often spelled "discs"), which are light on one side and dark on the other. Please see figure 1.

Players take turns placing disks on the board with their assigned color facing up. During a play, any disks of the opponent's color that are in a straight line and bounded by the disk just placed and another disk of the current player's color are turned over to the current player's color.

The object of the game is to have the majority of disks turned to display your color when the last playable empty square is filled.

You can refer to `http://www.tothello.com/html/guideline_of_reversed_othello.html` for more information of guideline, meanwhile, you can download the software to have a try from `http://www.tothello.com/html/download.html`. The game installer `tothello_trial_setup.exe` can also be found in the current folder.

2 Tasks

1. In order to reduce the complexity of the game, we think the board is 6×6.

2. There are several evaluation functions that involve many aspects, you can turn to `http://blog.sina.com.cn/s/blog_53ebdba00100cpy2.html` for help. In order to reduce the difficulty of the task, I have gaven you some hints of evaluation function in the file `Heuristic Function for Reversi (Othello).cpp`.

3. Please choose an appropriate evaluation function and use min-max and $\alpha - \beta$ prunning to implement the Othello game. The framework file you can refer to is `Othello.cpp`. Of course, I wish your program can beat the computer.

4. Write the related codes and take a screenshot of the running results in the file named E03_YourNumber.pdf, and send it to ai_2018@foxmail.com.

3 Codes

```
double max(double A, Do* choice)
{
    if(A > choice->score)
        return A;
    else
        return choice->score;
}

double min(double B, Do* choice)
{
    if(B < choice->score)
        return B;
    else
        return choice->score;
}


Do *Find(Othello *board, enum Option player, int step, int B, int A, Do *choice)
{
    if (step == 0)
    {
        choice->score = board->Judge(board);
        return choice;
    }
    Do* allChoices;
```

```
choice->score = -MAX;
choice->pos.first = -1;
choice->pos.second = -1;
choice->equal = false;

int num;
num = board->Rule(board, player);

if(num == 0)
{
    if(board->Rule(board, (enum Option)-player))
    {
        Othello tempBoard;
        Do nextChoice;
        Do *pNextChoice = &nextChoice;
        board->Copy(&tempBoard, board);
        pNextChoice =
        Find(&tempBoard, (enum Option) - player, step - 1, B, A, pNextChoice);
        choice->score = pNextChoice->score;
        return choice;
    }
    else
    {
        if (board->whiteNum < board->blackNum)
            choice->score = MAX;
        else if (board->whiteNum > board->blackNum)
            choice->score = MIN;
        else
            choice->equal = false;
        return choice;
    }
}
allChoices = (Do *)malloc(sizeof(Do) * num);
```

```
int k, i, j;
k = 0;
for (i = 0; i < 6; i++)
{
    for (j = 0; j < 6; j++)
    {
        if (board->cell[i][j].color ==
        SPACE && board->cell[i][j].stable)
        {
            allChoices[k].score = -MAX;
            allChoices[k].pos.first = i;
            allChoices[k].pos.second = j;
            k++;
        }
    }
}
if(player == BLACK)//max
{
    for (k = 0; k < num; k++)
    {
        Othello tempBoard;
        Do thisChoice, nextChoice;
        Do *pNextChoice = &nextChoice;
        thisChoice = allChoices[k];
        board->Copy(&tempBoard, board);
        board->Action(&tempBoard, &thisChoice, player);
        A = max(A, Find(&tempBoard, (enum Option) - player,
        step - 1, B, A, pNextChoice));
        if(B < A)
            break;
    }
    choice->score = A;
    choice->pos.first = allChoices[k%num].pos.first;
```

```
        choice->pos.second = allChoices[k%num].pos.second;
    }
    else//min
    {
        for (k = 0; k < num; k++)
        {
            Othello tempBoard;
            Do thisChoice, nextChoice;
            Do *pNextChoice = &nextChoice;
            thisChoice = allChoices[k];
            board->Copy(&tempBoard, board);
            board->Action(&tempBoard, &thisChoice, player);
            B = min(B, Find(&tempBoard, (enum Option) - player,
            step - 1, B, A, pNextChoice));
            if (B < A)
                break;
        }
        choice->score = B;
        choice->pos.first = allChoices[k%num].pos.first;
        choice->pos.second = allChoices[k%num].pos.second;
    }
    free(allChoices);
    return choice;
}

double Othello::Judge(Othello *board)
{
    int my_tiles = 0, opp_tiles = 0, i, j, k, my_front_tiles = 0, opp_front_tiles
    double p = 0, c = 0, l = 0, m = 0, f = 0, d = 0;

    int X1[] = {-1, -1, 0, 1, 1, 1};
    int Y1[] = {0, 1, 1, 1, 0, -1};
    int V[6][6] = { {20, -3, 11, 8, 8, 11},
```

```
                {-3, -7, -4, 1, 1, -4},
                {11, -4, 2, 2, 2, 2,},
                {8, 1, 2, -3, -3, 2},
                {8, 1, 2, -3, -3, 2},
                {11, -4, 2, 2, 2, 2} };

// Piece difference, frontier disks and disk squares
for (i = 0; i < 6; i++)
    for (j = 0; j < 6; j++)
    {
        if (board->cell[i][j].color == BLACK)
        {
            d += V[i][j];
            my_tiles++;
        }
        else if (board->cell[i][j].color == WHITE)
        {
            d -= V[i][j];
            opp_tiles++;
        }
        if (board->cell[i][j].color != SPACE)
        {
            for (k = 0; k < 6; k++)
            {
                x = i + X1[k];
                y = j + Y1[k];
                if (x >= 0 && x < 6 && y >= 0 && y < 6 && board->cell[x][y].co
                {
                    if (board->cell[i][j].color == BLACK)
                        my_front_tiles++;
                    else
                        opp_front_tiles++;
                    break;
```

```
                }
            }
        }
    }
if (my_tiles > opp_tiles)
    p = (100.0 * my_tiles) / (my_tiles + opp_tiles);
else if (my_tiles < opp_tiles)
    p = -(100.0 * opp_tiles) / (my_tiles + opp_tiles);
else
    p = 0;

if (my_front_tiles > opp_front_tiles)
    f = -(100.0 * my_front_tiles) / (my_front_tiles + opp_front_tiles);
else if (my_front_tiles < opp_front_tiles)
    f = (100.0 * opp_front_tiles) / (my_front_tiles + opp_front_tiles);
else
    f = 0;

// Corner occupancy
my_tiles = opp_tiles = 0;
if (board->cell[0][0].color == BLACK)
    my_tiles++;
else if (board->cell[0][0].color == WHITE)
    opp_tiles++;
if (board->cell[0][5].color == BLACK)
    my_tiles++;
else if (board->cell[0][5].color == WHITE)
    opp_tiles++;
if (board->cell[5][0].color == BLACK)
    my_tiles++;
else if (board->cell[5][0].color == WHITE)
    opp_tiles++;
if (board->cell[5][5].color == BLACK)
```

```
        my_tiles++;
    else if (board->cell[5][0].color == WHITE)
        opp_tiles++;
    c = 25 * (my_tiles - opp_tiles);

    // Corner closeness
    my_tiles = opp_tiles = 0;
    if (board->cell[0][0].color == SPACE)
    {
        if (board->cell[0][1].color == BLACK)
            my_tiles++;
        else if (board->cell[0][1].color == WHITE)
            opp_tiles++;
        if (board->cell[1][1].color == BLACK)
            my_tiles++;
        else if (board->cell[1][1].color == WHITE)
            opp_tiles++;
        if (board->cell[1][0].color == BLACK)
            my_tiles++;
        else if (board->cell[1][0].color == WHITE)
            opp_tiles++;
    }
    if (board->cell[0][5].color == SPACE)
    {
        if (board->cell[0][4].color == BLACK)
            my_tiles++;
        else if (board->cell[0][4].color == WHITE)
            opp_tiles++;
        if (board->cell[1][4].color == BLACK)
            my_tiles++;
        else if (board->cell[1][4].color == WHITE)
            opp_tiles++;
        if (board->cell[1][5].color == BLACK)
```

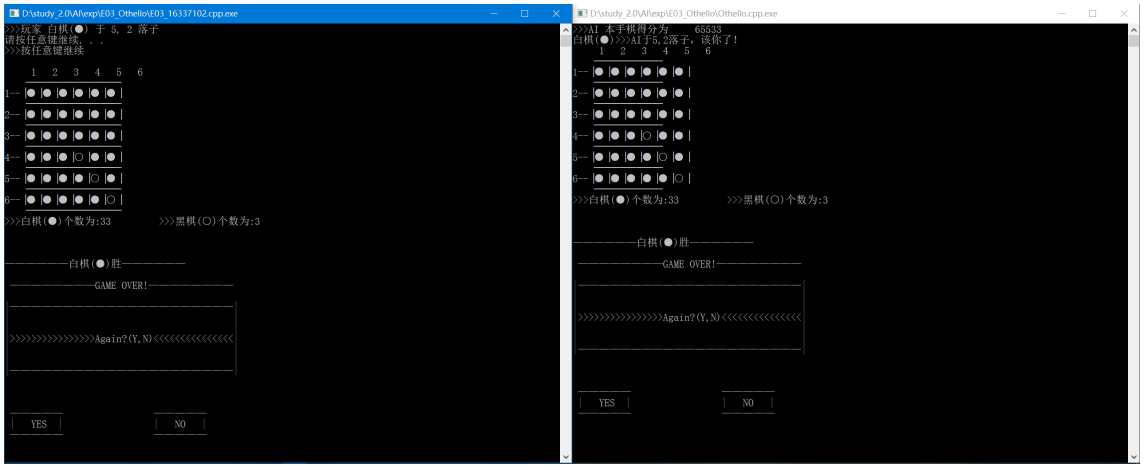
```
            my_tiles++;
        else if (board->cell[1][5].color == WHITE)
            opp_tiles++;
    }
    if (board->cell[5][0].color == SPACE)
    {
        if (board->cell[5][1].color == BLACK)
            my_tiles++;
        else if (board->cell[5][1].color == WHITE)
            opp_tiles++;
        if (board->cell[4][1].color == BLACK)
            my_tiles++;
        else if (board->cell[4][1].color == WHITE)
            opp_tiles++;
        if (board->cell[4][0].color == BLACK)
            my_tiles++;
        else if (board->cell[4][0].color == WHITE)
            opp_tiles++;
    }
    if (board->cell[5][5].color == SPACE)
    {
        if (board->cell[4][5].color == BLACK)
            my_tiles++;
        else if (board->cell[4][5].color == WHITE)
            opp_tiles++;
        if (board->cell[4][4].color == BLACK)
            my_tiles++;
        else if (board->cell[4][4].color == WHITE)
            opp_tiles++;
        if (board->cell[5][4].color == BLACK)
            my_tiles++;
        else if (board->cell[5][4].color == WHITE)
            opp_tiles++;
```

```
    }
    l = -12.5 * (my_tiles - opp_tiles);

    // Mobility
    my_tiles = board->Rule(board, BLACK);
    opp_tiles = board->Rule(board, WHITE);
    if (my_tiles > opp_tiles)
        m = (100.0 * my_tiles) / (my_tiles + opp_tiles);
    else if (my_tiles < opp_tiles)
        m = -(100.0 * opp_tiles) / (my_tiles + opp_tiles);
    else
        m = 0;

    // final weighted score
    double score = (10 * p) + (801.724 * c) + (382.026 * l) + (78.922 * m) + (74.3
    return score;
}
```

4 Results


D:\study_2.0\AI\exp\E03_Othello\E03_16337102.cpp.exe
>>>玩家 白棋(●) 于 5, 2 落子
请按任意键继续. . .
>>>按任意键继续
1 2 3 4 5 6
>>>白棋(●)个数为:33
>>>黑棋(○)个数为:3
白棋(●)胜
GAME OVER!
>>>>>>>>>>>>>>>Again?(Y,N)<<<<<<<<<<<<<<<
YES
NO
D:\study_2.0\AI\exp\E03_Othello\Othello.cpp.exe
>>>AI 本手棋得分为 65533
白棋(●)>>>AI于5,2落子，该你了！
1 2 3 4 5 6
>>>白棋(●)个数为:33
>>>黑棋(○)个数为:3
白棋(●)胜
GAME OVER!
>>>>>>>>>>>>>>>Again?(Y,N)<<<<<<<<<<<<<<<
YES
NO